L'Internet Software Consortium propose de nombreuses ressources pour **BIND**, un serveur de noms de domaine public très utilisé pour les stations UNIX [BIND 2002].

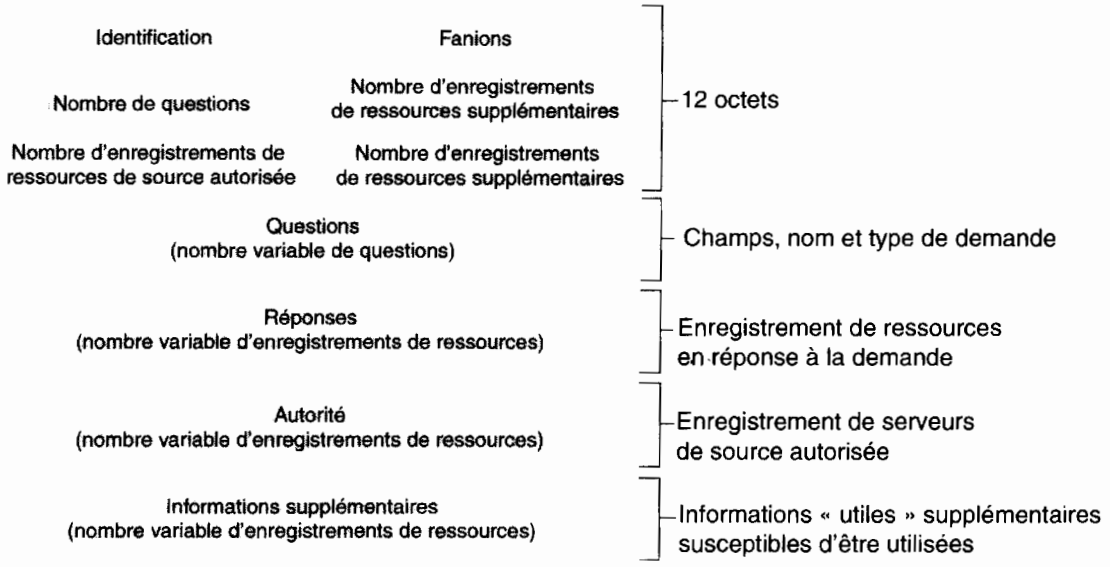

Figure 2.19 • Format de message DNS.

2.6 Programmation d'interface de connexion avec TCP

Cette section et les deux suivantes fournissent une introduction au développement d'applications de réseau. Le cœur d'une application de réseau se compose de deux logiciels, un logiciel client et un logiciel serveur (voir section 2.1). L'exécution de ces deux programmes génère un processus client et un processus serveur communiquant l'un avec l'autre par le biais d'interfaces de connexion. Pour créer une application, la principale fonction du développeur consiste à écrire le code de ces deux logiciels.

Il existe deux sortes d'applications client-serveur. L'une consiste en la mise en œuvre d'une norme de protocole définie dans un RFC. Pour cela, les logiciels client et serveur doivent se conformer à des règles dictées par ce RFC. Le logiciel client peut par exemple être une mise en œuvre du client FTP, défini dans le serveur 959 et le logiciel serveur une mise en œuvre du serveur FTP, défini dans le même RFC. Deux logiciels développés par deux programmeurs différents peuvent interfonctionner si les règles imposées par le RFC sont scrupuleusement respectées. En effet, la plupart des applications actuelles impliquent des échanges entre des logiciels client et serveur développés par des programmeurs différents (tels qu'un navigateur Netscape communiquant avec un serveur Web Apache ou le client FTP d'un PC chargeant un fichier sur un serveur FTP UNIX). Un logiciel client ou serveur ayant recours à un protocole défini dans un RFC doit utiliser l'accès correspondant à ce protocole. Les numéros d'accès sont évoqués à la section 2.1 et dans le prochain chapitre.

D'autres applications client-serveur sont des applications propriétaires, qui n'ont pas à se conformer à des normes officielles dans la mesure où le même développeur est responsable des deux parties et garde un contrôle intégral sur le code. Mais étant donné que le code ne met en œuvre aucun des protocoles référencés dans le domaine public, un développeur indépendant est dans l'incapacité de programmer un logiciel susceptible d'interfonctionner avec l'application. Lors du développement d'une application propriétaire, le responsable doit impérativement éviter d'utiliser l'un des numéros d'accès réservés pour les applications définies dans les RFC.

Nous allons maintenant passer en revue les aspects fondamentaux du développement d'une application client-serveur propriétaire. La première décision à prendre concerne le choix entre TCP et UDP. TCP procure un service orienté connexion ainsi qu'un canal de flux d'octets fiable faisant transiter les données entre deux systèmes terminaux. UDP, service sans connexion, envoie des paquets de données indépendants d'un système terminal à un autre sans fournir aucune garantie relative à leur bonne livraison.

Dans cette section, nous allons développer une application client simplifiée ayant recours à TCP, et dans la suivante, une application utilisant UDP, toutes deux en langage Java. Nous aurions aussi pu opter pour le C ou le C++, mais plusieurs raisons ont motivé notre choix. Tout d'abord, l'écriture d'une application est plus claire et plus propre en Java que dans n'importe quel autre langage. Le nombre de lignes de code est réduit et celles-ci sont accessibles à un programmeur débutant. De plus, la programmation client-serveur en Java a acquis une grande notoriété et pourrait devenir la norme dans un proche avenir. Les lecteurs intéressés par la programmation client serveur en C devraient trouver leur bonheur dans [Stevens 1997 ; Frost 1994 ; Kurose 1996].

2.6.1 Programmation d'interfaces de connexion avec TCP

La première section de ce chapitre décrit des processus opérant sur différents postes qui échangent des informations en envoyant des messages au travers des interfaces de connexions. Les premiers ont été comparés à des maisons et les secondes à des portes. Comme le montre la figure 2.20, l'interface de connexion (socket) constitue effectivement le sas séparant le processus d'application et TCP. Si le développeur d'applications contrôle tout ce qui se passe du côté de la couche d'application de cette interface, il a très peu d'autorité sur ce qui a lieu au niveau de la couche Transport. Tout au plus a-t-il la possibilité de définir quelques paramètres TCP, telle que la taille maximale de la mémoire tampon et des segments.

Examinons maintenant les échanges qui ont lieu entre les logiciels client et serveur. C'est au client qu'incombe la tâche de la prise de contact. Pour être en mesure de réagir au contact initial du client, le serveur doit être disponible. Ceci implique que le logiciel serveur soit lancé et exploitable en tant que processus. Il doit disposer d'une sorte de porte (l'interface de connexion) à laquelle le client vient « frapper ».

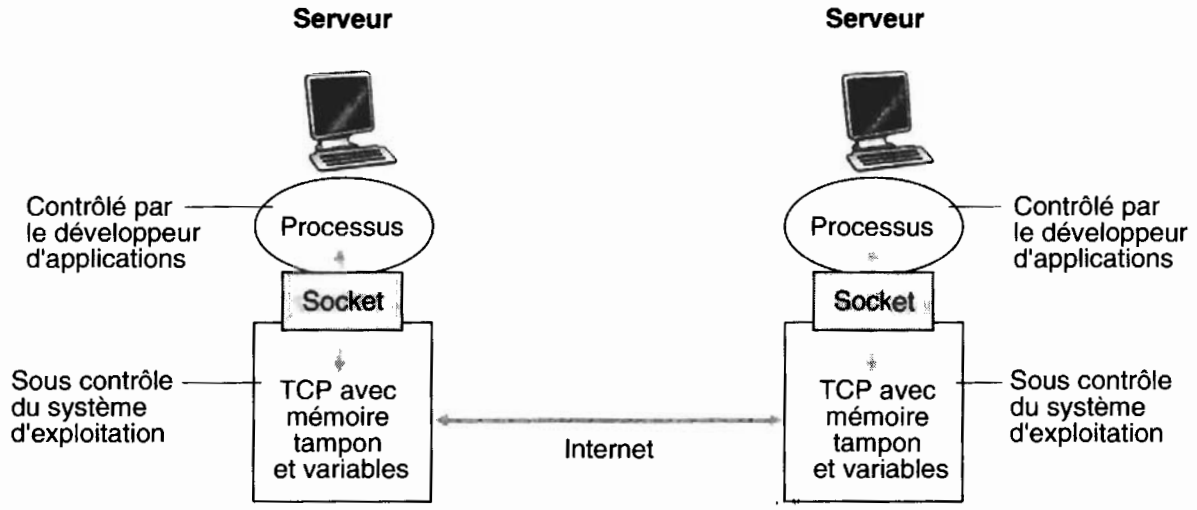

Figure 2.20 • Processus communiquant au travers d'interfaces de connexion (socket) TCP.

Une fois le processus serveur activé, le processus client peut établir une connexion TCP avec le serveur, ce qui se fait par la création d'une interface de connexion dans le logiciel client, au cours de laquelle ce dernier est invité à fournir l'adresse du processus serveur (adresse IP et numéro d'accès). Après quoi, TCP entame un échange de présentation à trois voies et établit la connexion TCP avec le serveur. Cette procédure de prise de contact est totalement transparente vis-à-vis des logiciels client et serveur.

Au cours de la procédure d'échange de présentation, le processus client frappe à la porte d'entrée du processus serveur. Lorsque le serveur « entend » les « coups » sur la porte, il génère une nouvelle voie d'accès (c'est-à-dire une nouvelle interface de connexion) dédiée spécialement à ce client. Dans l'exemple qui suit, la porte d'accueil est un objet `ServerSocket` appelé `welcomeSocket`. Lorsqu'un client frappe à cette porte, le logiciel fait appel à la méthode `accept()` du `welcomeSocket`, qui ordonne la création d'une nouvelle porte. À la fin des présentations, une connexion TCP relie l'**interface de connexion** du client à la nouvelle interface de connexion du serveur.

Du point de vue de l'application, la connexion TCP représente un conduit virtuel direct entre l'interface du client et l'interface de connexion du serveur. Le processus client peut envoyer n'importe quel type d'octets au travers de son interface. TCP garantit que le processus serveur les recevra dans le bon ordre (*via* l'interface de connexion). Qui plus est, de la même manière qu'une porte permet d'entrer et de sortir d'un bâtiment ou d'une pièce, les processus client et serveur peuvent respectivement recevoir et envoyer des octets au travers de leurs interfaces. Tout ceci est illustré à la figure 2.21.

Dans la mesure où les interfaces de connexion jouent un rôle fondamental dans les applications client-serveur, le développement de telles applications est également appelé « programmation d'interface de connexion ». Avant de passer à la programmation de notre application client-serveur, il convient de définir la notion de flux de données. Un **flux de données** (*stream*) est une séquence de caractères sortants ou entrants au cours ou à l'issue d'un processus. Il existe des **flux entrants**, et des **flux**

sortants issus de processus. Un flux entrant est associé à un périphérique d'entrée, tel que le clavier ou une interface de connexion laissant entrer des données en provenance de l'internet. Un flux sortant est associé à un périphérique de sortie, tel que l'écran ou une interface de connexion laissant passer des données en direction de l'internet.

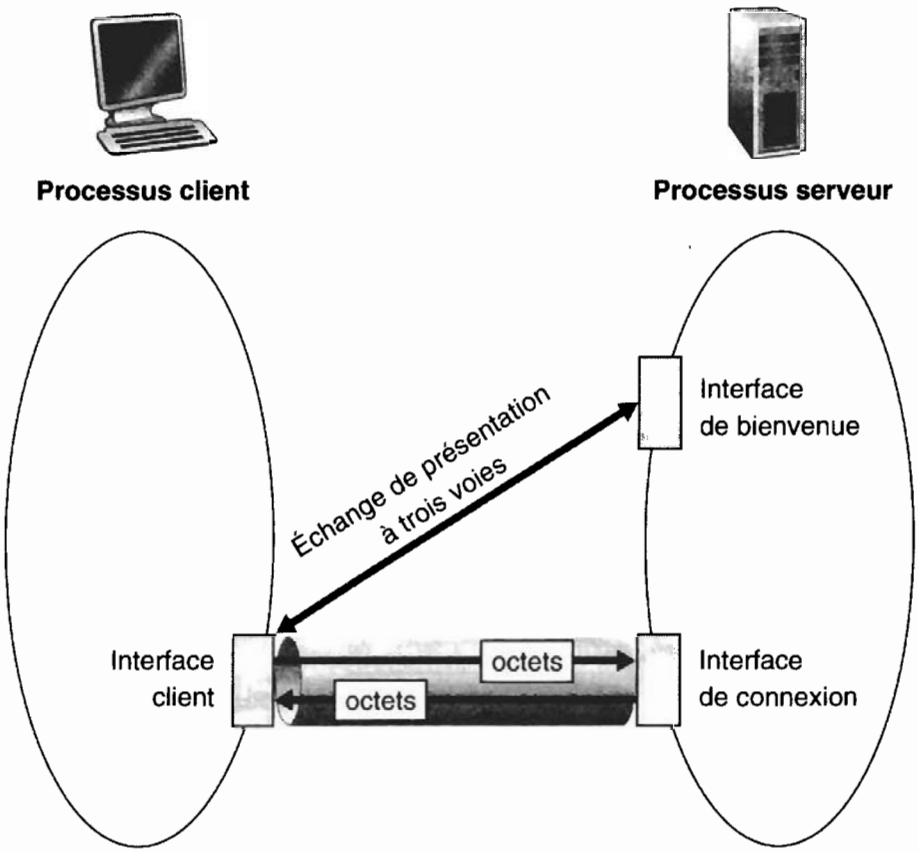

Figure 2.21 • Interface client, interface de bienvenue et interface de connexion.

2.6.2 Exemple d'application client-serveur en Java

Notre exemple de programmation d'interface de connexion (avec TCP et UDP) vise à développer une application dans le cadre de laquelle :

- Un client lit une ligne en provenance de son **périphérique d'entrée normalisé** (clavier) et envoie cette ligne au serveur *via* son interface.
- Le serveur lit la ligne reçue au travers de son interface de connexion.
- Le serveur convertit la ligne en majuscules.
- Le serveur envoie la ligne modifiée au client *via* son interface de connexion.
- Le client lit la ligne modifiée reçue au travers de son interface et l'affiche sur son **périphérique de sortie normalisé (écran)**.

Commençons par le cas d'un client et d'un serveur communiquant au moyen d'un service de transport orienté connexion (TCP). La figure 2.22 illustre les principaux échanges entre les deux interfaces.

Ensuite nous nous intéresserons au couple de logiciels client-serveur pour une mise en œuvre de l'application sous TCP. À la fin de chaque programme, nous décrivons chaque étape ligne par ligne. Le logiciel client s'appelle `TCPClient.java` et le logiciel serveur `TCPServer.java`. Afin de faire ressortir les aspects fondamentaux de ces programmes, le code présenté ici a volontairement été épuré, ce qui le rend convenable, mais pas encore infaillible du point de vue de sa robustesse. Un « bon » code comprendrait de toute évidence quelques lignes en plus.

Une fois les deux logiciels compilés sur leurs serveurs respectifs, le logiciel serveur est d'abord lancé sur le serveur, ce qui génère un processus serveur. Comme indiqué ci-dessus, ce processus se met alors en attente d'une prise de contact par un processus client généré sur lancement du logiciel client par le terminal de l'utilisateur. Une fois cette prise de contact effectuée, une connexion TCP s'établit entre les deux entités. Le client peut ensuite « utiliser » l'application pour envoyer une ligne et attendre une version enrichie de la part du logiciel serveur.

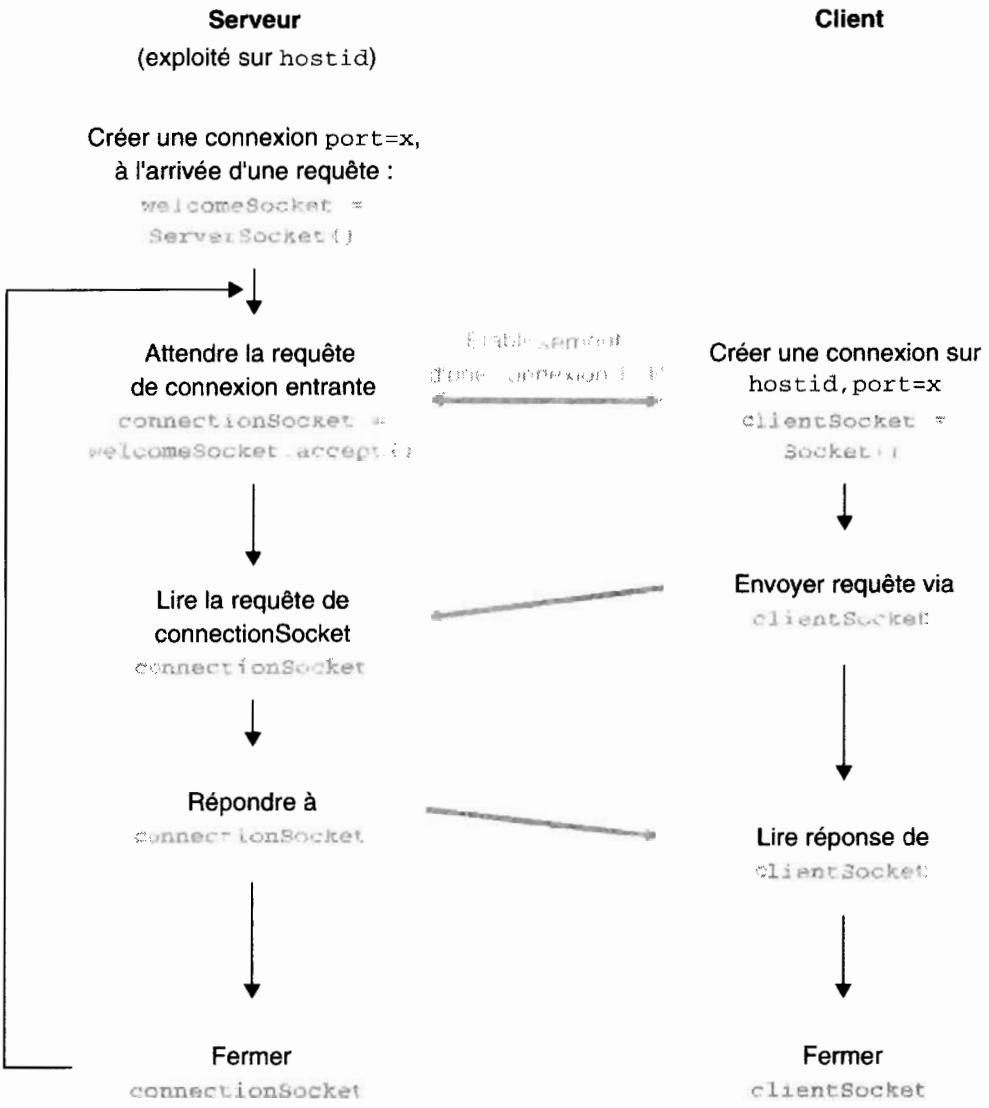

Figure 2.22 • L'application client-serveur, dans le cadre d'un service de transport orienté connexion.

TCPClient.java

Voici le code correspondant au pôle client de l'application :

```
import java.io.*;
import java.net.*;
class TCPClient {
   public static void main(String argv[]) throws Exception
   {
      String sentence;
      String modifiedSentence;
      BufferedReader inFromUser =
         new BufferedReader(
               new InputStreamReader(System.in));
      Socket clientSocket = new Socket("hostname", 6789);
      DataOutputStream outToServer =
         new DataOutputStream(
               clientSocket.getOutputStream());
      BufferedReader inFromServer =
         new BufferedReader(new InputStreamReader(
               clientSocket.getInputStream()));
      sentence = inFromUser.readLine();
      outToServer.writeBytes(sentence + '\n');
      modifiedSentence = inFromServer.readLine();
      System.out.println("FROM SERVER: " +
                              modifiedSentence);
      clientSocket.close();
   }
}
```

Le logiciel TCPClient crée trois flux de données et une interface de connexion, comme indiqué à la figure 2.23.

L'interface de connexion est appelée `clientSocket`. Le flux de données `inFromUser` est un flux entrant pour le logiciel, donc associé au périphérique d'entrée normalisé (le clavier). Lorsque l'utilisateur pianote sur son clavier, les caractères rejoignent le flux de données `inFromUser`. Le flux de données `inFromServer` est un autre flux entrant pour le logiciel, associé cette fois à l'interface de connexion. C'est ce flux qu'utilisent les caractères en provenance du réseau. Enfin, le flux de données `outToServer` est un flux sortant du logiciel, également associé à l'interface de connexion. Les caractères que le client envoie à destination du réseau rejoignent le flux de données `outToServer`.

Intéressons-nous maintenant aux différentes lignes de code :

```
import java.io.*;
import java.net.*;
```

Les noms `java.io` et `java.net` font référence à des paquetages Java. Le paquetage `java.io` contient différentes classes pour les flux sortants et entrants. Plus précisément, `java.io` contient les classes `BufferedReader` et `DataOutputStream`, utilisées par le logiciel pour créer les trois types de flux de données présentés ci-dessus.

`java.net` fournit des classes pour le support réseau, à savoir les classes `Socket` et `ServerSocket`. L'objet `clientSocket` de ce logiciel provient de la classe `Socket`.

```
class TCPClient {
   public static void main(String argv[]) throws Exception
      {.....}
}
```

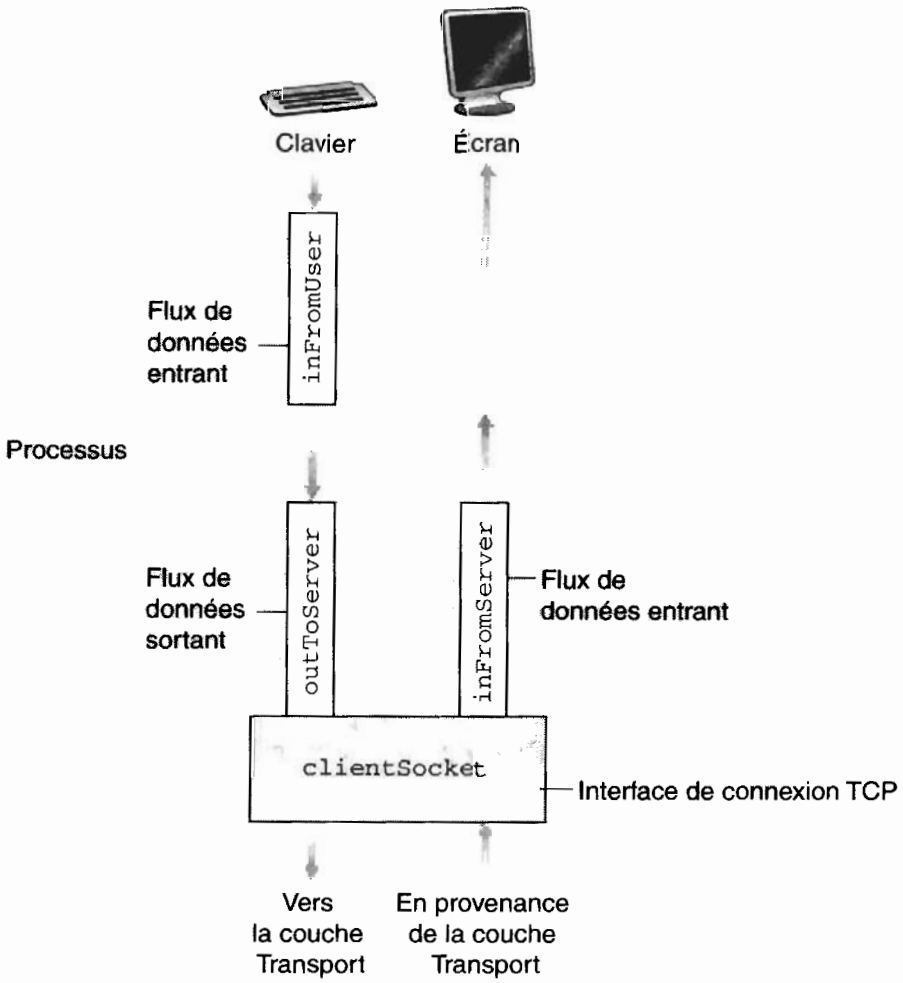

Figure 2.23 • Le logiciel TCPClient dispose de trois flux différents pour la circulation des caractères.

Jusqu'à présent, nous n'avons vu que du code élémentaire, soit celui que vous trouverez au début de tout programme Java. La première ligne correspond au début d'un bloc de définition de classe. Le mot clé `class` débute la définition de classe pour la classe appelée `TCPClient`. Une classe contient des variables et des méthodes, insérées entre des accolades droites ou gauches qui ouvrent et ferment le bloc de définition de classe. La catégorie `TCPClient` n'a pas de variable de classe et elle ne compte qu'une seule méthode, la méthode `main()`. Les méthodes du langage Java sont analogues aux fonctions et aux procédures d'autres langages de programmation, tel que le C. La méthode « main » en Java est comparable à la fonction « main » en C et en C++. Lorsque l'interpréteur Java exécute une application (à la demande se rapportant à la

classe de contrôle de l'application), il appelle tout d'abord la méthode « main » de la classe. Celle-ci appelle ensuite toutes les autres méthodes nécessaires au fonctionnement de l'application. Pour cette introduction à la programmation d'interface de connexion en Java, nous n'expliquerons pas les mots clés `public`, `static`, `void`, `main` et `throws Exceptions` (bien qu'il soit nécessaire de les inclure dans le code).

```
String sentence;
String modifiedSentence;
```

Les deux lignes précédentes font référence à des objets de type `String`. L'objet `sentence` est la chaîne de données entrée par l'utilisateur qui est envoyée au serveur. L'objet `modifiedSentence` est la chaîne de données reçue du serveur qui s'affiche sur l'écran de l'utilisateur.

```
BufferedReader inFromUser =
   new BufferedReader (new InputStreamReader(System.in)) ;
```

Les deux lignes précédentes génèrent l'objet de flux de données `inFromUser` de type `Buffered Reader`. Le flux entrant est initialisé au moyen de l'argument `System.in`, qui attache le flux au périphérique d'entrée normalisé. La commande permet au client de lire du texte à partir de son clavier.

```
Socket clientSocket = new Socket("hostname", 6789);
```

Cette ligne génère l'objet `clientSocket` de type `Socket`. Elle conduit également à l'établissement de la connexion TCP entre le client et le serveur. La chaîne de données `"hostname"` doit être remplacée par le nom du serveur (par exemple, `"fling.seas.upenn.edu"`). Avant établissement de la connexion TCP, le client effectue une recherche DNS pour obtenir l'adresse IP du serveur. Le nombre 6789 correspond au numéro d'accès. Il est possible d'utiliser un autre numéro d'accès, mais après s'être assuré que la même référence sera utilisée par l'autre pôle de l'application. Comme vu précédemment, l'adresse IP du serveur et le numéro d'accès de l'application sont les deux identifiants du processus serveur.

```
DataOutputStream outToServer =
   new DataOutputStream(clientSocket.getOutputStream());
BufferedReader inFromServer =
   new BufferedReader(new inputStreamReader(
                     clientSocket.getInputStream()));
```

Ces lignes génèrent des objets de flux de données associés à l'interface de connexion. Le flux de données `outToServer` déclenche le processus de sortie du flux vers l'interface. Le flux de données `inFrom-Server` ouvre le processus d'entrée en provenance de l'interface (voir figure 2.23).

```
sentence = inFromUser.readLine();
```

Cette formulation place une ligne de caractères composée par l'utilisateur au sein de la chaîne de flux de données `sentence`. Celle-ci continue de rassembler les caractères jusqu'à ce que l'utilisateur termine sa ligne au moyen d'un retour chariot.

Du périphérique d'entrée, celle-ci rejoint alors le flux de données `inFromUser` vers la chaîne de données `sentence`.

```
outToServer.writeBytes(sentence + '\n');
```

Cette ligne insère la chaîne de données `sentence` accompagnée d'un retour chariot dans le flux de données `outToServer`. La phrase composée circule au travers de l'interface de connexion du client en direction du conduit TCP. Le client se met ensuite en attente de caractères en provenance du serveur.

```
modifiedSentence = inFromServer.readLine();
```

Lorsqu'un groupe de caractères arrive en provenance du serveur, il traverse le flux de données `inFromServer` et il est intégré à la chaîne de données `modifiedSentence`. Les caractères continuent à s'y accumuler jusqu'à la fin de la ligne qui se termine par le signe du retour chariot.

```
System.out.println("FROM SERVER " + modifiedSentence);
```

Cette ligne affiche à l'écran la chaîne de données `modifiedSentence` renvoyée par le serveur.

```
clientSocket.close();
```

Cette dernière ligne ferme l'interface de connexion et interrompt donc la connexion TCP entre le client et le serveur. Elle provoque l'envoi d'un message du TCP client vers le TCP serveur (voir section 3.5).

TCPServer.java

Intéressons-nous maintenant au logiciel serveur :

```
import java.io.*;
import java.net.*;
class TCPServer {
   public static void main(String argv[]) throws Exception
      {
         String clientSentence;
         String capitalizedSentence;
         ServerSocket welcomeSocket = new ServerSocket
            (6789);
         while(true) {
            Socket connectionSocket = welcomeSocket.
               accept();
            BufferedReader inFromClient =
               new BufferedReader(new InputStreamReader(
                  connectionSocket.getInputStream()));
            DataOutputStream outToClient =
               new DataOutputStream(
                  connectionSocket.getOutputStream());
            clientSentence = inFromClient.readLine();
            capitalizedSentence =
                  clientSentence.toUpperCase() + '\n';
```

```
                outToClient.writeBytes(capitalizedSentence);
            }
        }
}
```

Le logiciel `TCPServer` a de nombreux points communs avec le `TCPClient`. Dans l'analyse des lignes de code du `TCPServer.java` qui suivent, nous ne commenterons que les commandes se distinguant de celles du `TCPClient.java`.

La première ligne du `TCPServer` réellement différente du code du `TCPClient` est la suivante :

```
ServerSocket welcomeSocket = new ServerSocket(6789);
```

Cette ligne génère l'objet `welcomeSocket`, qui est du type `ServerSocket`. Le `welcomeSocket` est une sorte de porte qui attend qu'un client frappe pour s'ouvrir (sur l'accès 6789). La ligne suivante est :

```
Socket connectionSocket = welcomeSocket.accept();
```

Cette ligne génère une nouvelle interface de connexion, intitulée `connectionSocket` (dont le numéro d'accès est identique), qui est créée lorsqu'un client frappe à la porte du `welcomeSocket`. La raison pour laquelle les deux interfaces partagent le même numéro d'accès est expliquée au chapitre 3. TCP établit ensuite un conduit virtuel direct entre le `clientSocket` au niveau du client et le `connectionSocket` au niveau du serveur. Le client et le serveur peuvent dès lors s'échanger des octets de données, qui arrivent à destination dans le bon ordre. Une fois le `connectionSocket` établi, le serveur peut rester attentif à des requêtes d'autres clients pour l'application ayant recours au `welcomeSocket`. La version du programme retranscrite ici ne présente pas cette fonction, mais quelques lignes suffiraient pour l'en doter. Le programme génère ensuite différents objets de flux de données, comparables à ceux du `clientSocket`. Considérez maintenant la commande suivante :

```
capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

Cette commande représente le cœur même de l'application. Elle prend en effet la ligne envoyée par le client, l'enrichit et lui ajoute un retour chariot. Dans ce but, elle a recours à la méthode `toUpperCase()`. Toutes les autres commandes sont d'ordre périphérique, et sont impliquées dans la communication avec le client.

Pour tester le couple formé par ces deux logiciels, il suffit d'installer et de compiler le `TCPClient.java` et le `TCPServer.java` sur deux serveurs différents, en faisant bien attention à inclure le nom du serveur dans le `TCPClient.java`. Lancez ensuite le programme `TCPServer.class`, le programme serveur compilé sur le serveur. Ceci génère un processus serveur qui se met en attente d'une requête client. Lancez alors `TCPClient.class`, le programme client compilé, sur le poste client. Ceci génère un processus client et établit une connexion TCP avec le processus serveur. Enfin, pour utiliser l'application, entrez une ligne de texte terminée par un retour chariot.

Pour développer votre propre application client-serveur, vous pouvez commencer par apporter de légères modifications aux deux programmes présentés ici. Au lieu de convertir toutes les lettres en majuscules, le serveur pourrait par exemple recenser le nombre de fois qu'apparaît la lettre « s » et renvoyer un message indiquant le nombre d'occurences.

2.7 Programmation de l'interface de connexion en UDP

Dans la section précédente, nous avons appris que deux processus communiquant l'un avec l'autre *via* TCP passaient par une sorte de conduit, qui persiste jusqu'à ce que l'un d'entre eux le ferme de façon délibérée. Lorsque l'un des processus souhaite envoyer des octets à l'autre, il lui suffit de les insérer dans le conduit. Le processus expéditeur n'a pas à préciser d'adresse de destination, puisque le conduit est en connexion logique avec le destinataire. Qui plus est, ce conduit procure un canal de flux d'octets fiable, ce qui signifie que les octets atteignent le processus destinataire dans l'ordre dans lequel ils ont été envoyés.

UDP permet également à deux processus (ou plus) exploités sur des serveurs différents de communiquer, à ceci près que UDP est un service sans connexion qui ne procède pas à l'échange initial de présentations donnant lieu à l'établissement d'un conduit entre les deux processus. À défaut de bénéficier d'un conduit direct, un processus souhaitant envoyer un groupe d'octets à un autre doit systématiquement préciser à chaque fois l'adresse du destinataire, ce qui rend UDP comparable à un service de taxi. L'adresse de destination est une ligne d'octets constituée de l'adresse IP du destinataire et du numéro d'accès du processus de destination. Nous appellons « paquets » les groupes d'octets d'information associés à une adresse IP et à un numéro d'accès.

Après avoir généré un « paquet », le processus expéditeur le place sur le réseau au travers de son interface de connexion. De l'autre côté de cette interface se trouve un « taxi » prêt à emmener ce paquet à sa destination. Mais celui-ci n'offre aucune garantie relative à l'arrivée du paquet à la bonne destination (une panne peut se produire à tout moment). En d'autres termes, UDP procure un service de transport non fiable à ses processus de communication mais il ne garantit pas que les datagrammes qu'il prend en charge arriveront à bon port.

Dans cette section, nous illustrons la programmation client-serveur en UDP en reprenant l'application précédente. Malgré leurs ressemblances, le code Java utilisé pour UDP diffère fondamentalement du code employé pour TCP. En l'occurrence, (1) aucun échange de présentation n'a lieu entre les deux processus, qui ne génèrent donc pas de d'interface de bienvenue, (2) aucun flux de données n'est associé aux interfaces de connexion, (3) le serveur expéditeur génère des paquets en attachant l'adresse IP de destination et le numéro d'accès à chaque groupe d'octets qu'il envoie

et (4) le processus récepteur doit ouvrir chaque paquet qu'il reçoit pour accéder à son contenu. Voici le principe de fonctionnement de notre application :

Un client lit une ligne de texte en provenance de son périphérique d'entrée (son clavier) et il envoie cette ligne au serveur *via* son interface.

1. Le serveur lit la ligne reçue au travers de son interface de connexion.
2. Le serveur convertit la ligne en majuscules.
3. Le serveur envoie la ligne modifiée au client *via* son interface de connexion.
4. Le terminal du client lit la ligne modifiée reçue au travers de son interface et l'affiche sur son périphérique de sortie (son écran).

La figure 2.24 présente les principales actions effectuées par un client et un serveur communiquant au moyen d'un service de transport sans connexion (UDP).

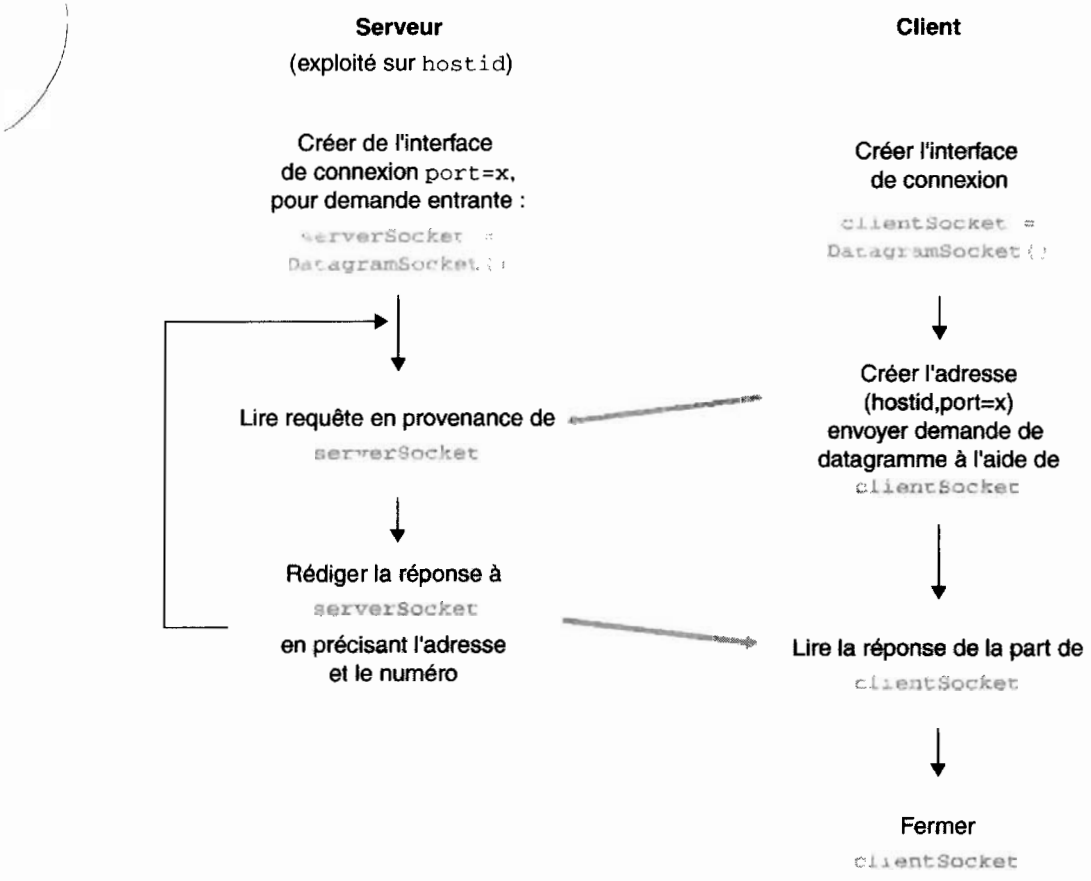

Figure 2.24 • L'application client-serveur associée à un service de transport sans connexion.

UDPClient.java

Voici le code correspondant au pôle client de l'application :

```
import java.io.*;
import java.net.*;
class UDPClient {
   public static void main(String args[]) throws Exception
```

```
    {
       BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader
                      (System.in));
       DatagramSocket clientSocket = new DatagramSocket();
       InetAddress IPAddress =
                      InetAddress.getByName("hostname");
       byte[] sendData = new byte[1024];
       byte[] receiveData = new byte[1024];
       String sentence = inFromUser.readLine();
       sendData = sentence.getBytes();
       DatagramPacket sendPacket =
          new DatagramPacket(sendData, sendData.length,
                      IPAddress, 9876);
       clientSocket.send(sendPacket);
      DatagramPacket receivePacket =
         new DatagramPacket(receiveData,
                    receiveData.length);
      clientSocket.receive(receivePacket);
      String modifiedSentence =
         new String(receivePacket.getData());
      System.out.println("FROM SERVER:" +
                 modifiedSentence);
      clientSocket.close();
   }
}
```

Comme indiqué à la figure 2.25, le programme `UDPClient.java` génère un flux de données et une interface de connexion. Cette interface, de type `Datagram-Socket`, est appelée `clientSocket`. Au niveau du client, le protocole UDP utilise un type d'interface de connexion différent de TCP, en l'occurrence, un `DatagramSocket` au lieu d'un `Socket`. Le flux de données `inFromUser` est un flux entrant relié au périphérique d'entrée, c'est-à-dire au clavier. Le flux de données est similaire à celui de la version TCP du programme (les caractères entrés au moyen du clavier rejoignent le flux de données `inFromUser`), à ceci près qu'aucun flux (entrant ou sortant) n'est associé à l'interface de connexion. Plutôt que d'envoyer des flux de données associés à un objet `Socket`, UDP véhicule en effet des paquets individuels au travers de l'objet `DatagramSocket`.

Examinons maintenant les parties du code qui diffèrent sensiblement de celles du programme `TCPClient.java`.

```
DatagramSocket clientSocket = new DatagramSocket () ;
```

Cette ligne génère un objet `clientSocket` de type `DatagramSocket`.

Contrairement au programme `TCPClient.java`, cette ligne de code n'aboutit pas à l'établissement d'une connexion TCP. En l'occurrence, l'exécution de cette ligne n'incite pas le client à prendre contact avec le serveur. Par conséquent, `Datagram-Socket()` ne prend pas en compte le nom du serveur ou son numéro d'accès comme des arguments. Pour reprende l'analogie précédente, cette ligne de code génère bien une porte pour le processus client, mais n'établit aucun conduit entre les deux processus.

```
InetAddress IPAddress = InetAddress.getByName ("hostname") ;
```

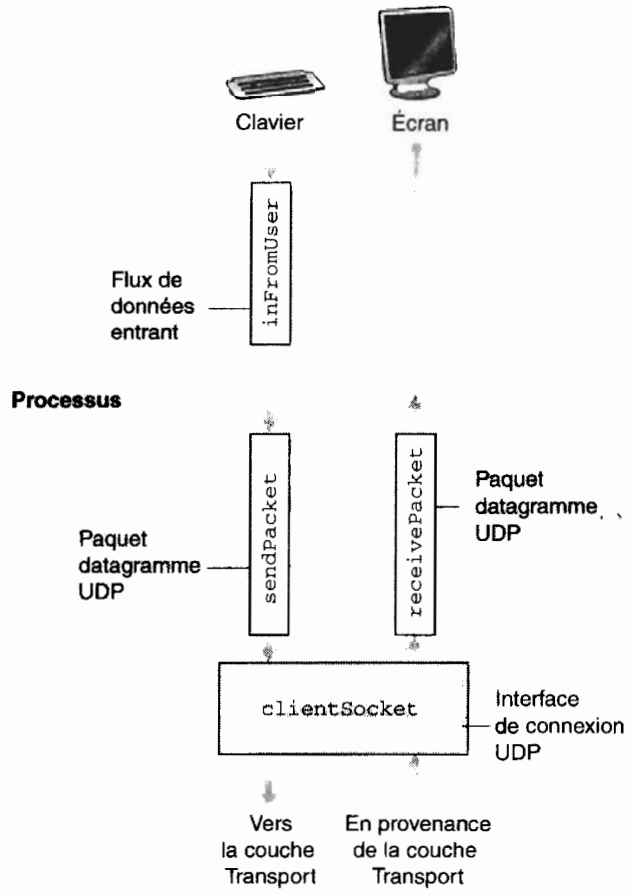

Figure 2.25 • `UDPClient` ne génère qu'un seul flux de données. L'interface accepte et envoie des paquets *via* le processus

Pour envoyer des octets à un processus de destination, il faut connaître son adresse, sachant que l'adresse IP représente une partie de cette adresse. La ligne précédente constitue un appel à conversion du nom de serveur en adresse IP. Dans cet exemple, il est inséré dans le code par le développeur. La version TCP du client avait également recours au DNS, mais de manière implicite (plutôt qu'explicite). La méthode `getByName()` utilise le nom du serveur et le convertit en adresse IP, qu'elle place ensuite au sein de l'objet `IPAddress` de type `InetAddress`.

```
byte [] sendData = new byte[1024];
byte [] receiveData = new byte [1024];
```

Les champs `sendData` et `receiveData` comprennent respectivement les données envoyées et reçues par le client.

```
sendData = sentence.getBytes ();
```

Cette ligne de code effectue essentiellement une conversion de type, prenant la phrase sous forme de chaîne de caractères dans le flux de données et la renommant `sendData`, ce qui constitue alors un tableau d'octets.

```
DatagramPacket sendPacket =
      New DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Cette ligne de code génère le paquet `sendPacket` que le client charge sur le réseau au travers de son interface de connexion. Ce paquet contient les données à envoyer, `sendData`, accompagnées d'informations relatives à leur longueur, de l'adresse IP du

serveur et du numéro d'accès de l'application (9876 dans notre exemple). Notez que `sendPacket` est de type `DatagramPacket`.

```
clientSocket.send(sendPacket);
```

Dans cette ligne, `send()` appliqué à l'objet `clientSocket` prend le paquet qui vient d'être créé et le charge sur le réseau à travers `clientSocket`. Vous remarquez que UDP envoie la ligne de caractères d'une manière très différente de TCP, qui lui se contente d'intégrer la chaîne de caractères à un flux de données en connexion logique avec le serveur. Après l'envoi du paquet, le client attend une réponse de la part du serveur, arrivant sous la forme d'un autre paquet.

```
DatagramPacket receivePacket =
new DatagramPacket(receiveData, receiveData.length);
```

Au moyen de la ligne précédente, alors qu'il est en attente d'une réponse de la part du serveur, le client génère un emplacement pour `receivePacket`, qui est un objet de type `Datagram-Packet`.

```
clientSocket.receive(receivePacket);
```

Puis le client se met en attente jusqu'à la réception du paquet de réponse. Une fois la réponse parvenue, il place le paquet reçu dans `receivePacket`.

```
String modifiedSentence =
new String(receivePacket.getData());
```

La ligne précédente sert à extraire les données de `receivePacket` et à effectuer une conversion de type, qui traduit un tableau d'octets en chaîne `modifiedSentence`.

```
System.out.println("FROM SERVER:" + modifiedSentence);
```

Cette ligne, identique à celle présente dans `TCPClient`, affiche la chaîne `modified-Sentence` sur l'écran du client.

```
clientSocket.close();
```

Cette dernière ligne ferme l'interface de connexion. Comme UDP procure un service sans connexion, cette ligne ne conduit pas le client à envoyer un message de couche Transport au serveur (à la différence de ce qui a lieu à la fin de `TCPClient`).

UDPServer.java

Voyons maintenant à quoi ressemble le pôle client de l'application :

```
import java.io.*;
import java.net.*;
class UDPServer {
   public static void main(String args[]) throws Exception
      {
     DatagramSocket serverSocket = new
                          DatagramSocket(9876);
```

```
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];
        while(true)
           {
                DatagramPacket receivePacket =
                   new DatagramPacket(receiveData,
                            receiveData.length);
                serverSocket.receive(receivePacket);
                String sentence = new String(
                            receivePacket.getData());
                InetAddress IPAddress =
                            receivePacket.getAddress();
                int port = receivePacket.getPort();
                String capitalizedSentence =
                            sentence.toUpperCase();
                sendData = capitalizedSentence.getBytes();
                DatagramPacket sendPacket =
                new DatagramPacket(sendData,
                         sendData.length, IPAddress, port);
                serverSocket.send(sendPacket);
           }
        }
}
```

La figure 2.26 illustre la création d'une interface de connexion (appelée `serverSocket`) par le programme `UDPServer.java`. Il s'agit d'un objet de type `DatagramSocket`, tout comme l'interface située au niveau du pôle client de l'application. Ici non plus, aucun flux de données n'est associé à l'interface.

Intéressons-nous maintenant aux lignes de code se distinguant de celles du `TCPServer.java`.

```
DatagramSocket serverSocket = new DatagramSocket(9876);
```

Cette ligne génère `DatagramSocket serverSocket` sur l'accès 9876. Toutes les données envoyées et reçues passent au travers de cette interface. Comme UDP procure un service sans connexion, il n'est pas nécessaire de mettre en place une nouvelle interface et de rester à l'écoute de nouvelles requêtes de connexion, comme avec `TCPServer.java`. Tous les clients accédant à cette application envoient leurs paquets au travers de cette seule porte `serverSocket`.

```
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
```

Les trois lignes de code précédentes ont pour fonction d'ouvrir le paquet en provenance du client. La première extrait les données qu'il contient et elle les place dans `String sentence`. (Il existe une ligne analogue dans `UDPClient`). La deuxième extrait l'adresse IP et la troisième le numéro d'accès du client, choisi par le client et différent du numéro d'accès du serveur (9876). (Les numéros d'accès client seront traités au prochain chapitre.)

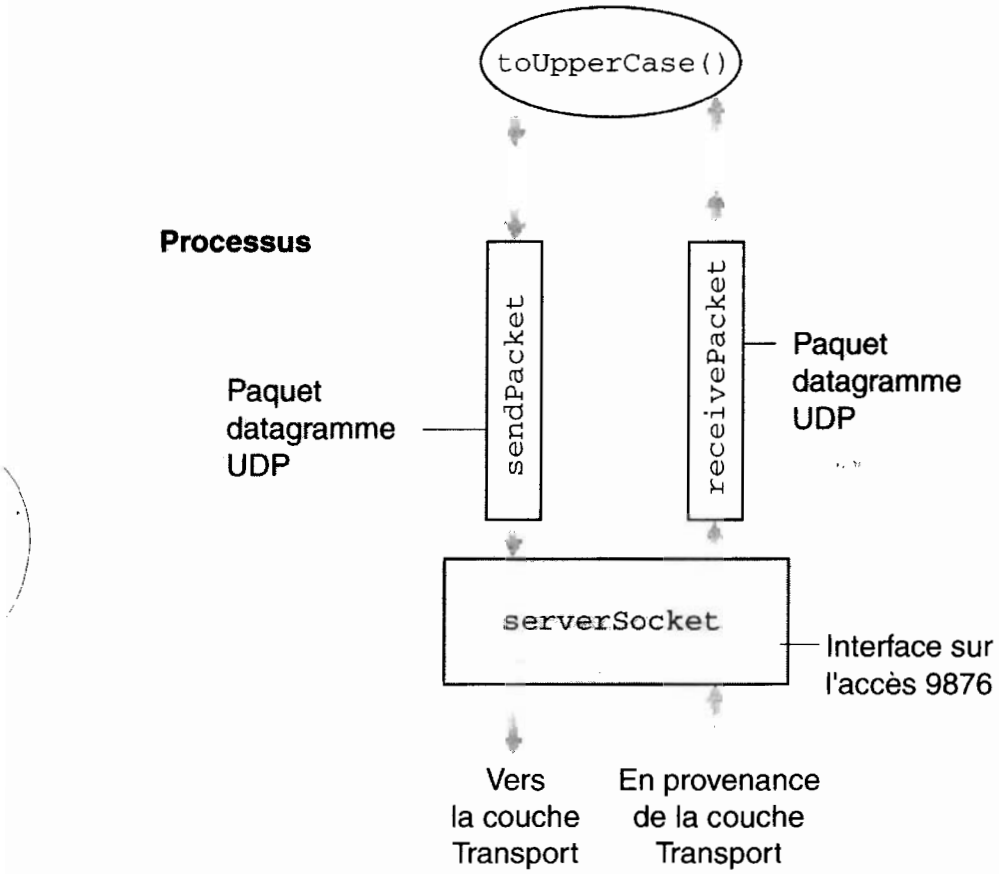

Figure 2.26 • `UDPServer` ne comprend aucun flux de données. L'interface de connexion accepte et envoie des paquets *via* le processus.

Afin de pouvoir renvoyer la phrase en capitale d'imprimerie au client, le serveur doit impérativement obtenir ses coordonnées (adresse IP ou numéro d'accès).

Ceci complète l'analyse du couple de programmes UDP. Pour tester cette application, il suffit d'installer et de compiler les programmes `UDPClient.java` et `UDPServer.java` sur deux serveurs différents. (Faites bien attention au nom de serveur donné à `UDPClient.java`). Lancez ensuite les deux programmes sur leur serveur respectif sachant que, contrairement à TCP, il est possible d'exécuter le pôle client avant de lancer le pôle serveur. En effet, le processus client n'essaie pas d'établir de connexion avec le serveur lorsque le pôle client est exécuté. Une fois que les deux programmes sont lancés, vous pouvez utiliser l'application en entrant une ligne sur le poste client.

2.8 Mise en place d'un serveur Web simplifié

Maintenant que HTTP n'a plus de secret pour vous et que vous savez programmer des applications client-serveur en Java, tentons de mettre toutes ces notions en pratique pour concevoir un petit serveur Web. Contrairement à ce que l'on pourrait croire, il s'agit presque d'un jeu d'enfant.

2.8.1 Fonctions du serveur Web

Dans cette section, nous allons essayer de mettre en place un serveur doté des fonctions suivantes :

- Traitement d'une seule requête HTTP à la fois.
- Prise en charge et analyse de la requête http.
- Obtention du fichier recherché à partir du système de fichiers du serveur.
- Création d'un message de réponse HTTP constitué du fichier sollicité précédé par un en-tête.
- Envoi d'une réponse directement au client.

Par souci de clarté, nous nous en tiendrons au code le plus simple possible, ce qui signifie qu'il sera loin d'être complet ! Nous éviterons par exemple toutes les exceptions et partirons du principe que le client sollicite un objet contenu dans le système de fichiers du serveur.

WebServer.java

Voici le code d'un serveur Web simplifié :

```
import java.io.*;
import java.net.*;
import java.util.*;
class WebServer {
   public static void main(String argv[]) throws Exception
   {
      String requestMessageLine;
      String fileName;
      ServerSocket listenSocket = new ServerSocket(6789);
      Socket connectionSocket = listenSocket.accept();
      BufferedReader inFromClient =
         new BufferedReader(new InputStreamReader(
               connectionSocket.getInputStream()));
      DataOutputStream outToClient =
         new DataOutputStream(
               connectionSocket.getOutputStream());
      requestMessageLine = inFromClient.readLine();
      StringTokenizer tokenizedLine =
         new StringTokenizer(requestMessageLine);
      if (tokenizedLine.nextToken().equals("GET")){
         fileName = tokenizedLine.nextToken();
         if (fileName.startsWith("/") == true )
               fileName = fileName.substring(1);
         File file = new File(fileName);
         int numOfBytes = (int) file.length();
         FileInputStream inFile = new FileInputStream (
               fileName);
         byte[] fileInBytes = new byte[numOfBytes];
         inFile.read(fileInBytes);
```

```
            outToClient.writeBytes(
                  "HTTP/1.0 200 Document Follows\r\n");
            if (fileName.endsWith(".jpg"))
               outToClient.writeBytes("Content-Type:
                     image/jpeg\r\n");
            if (fileName.endsWith(".gif"))
               outToClient.writeBytes("Content-Type:
                     image/gif\r\n");
            outToClient.writeBytes("Content-Length: " +
                  numOfBytes + "\r\n");
            outToClient.writeBytes("\r\n");
            outToClient.write(fileInBytes, 0, numOfBytes);
            connectionSocket.close();
            }
         else System.out.println("Bad Request Message");
         }
}
```

Analysons ce programme. La première moitié du code est semblable à celle de `TCP-Server.java`. En effet, tout comme avec ce dernier, nous importons `java.io` et `java.net`. Puis vient s'ajouter `java.util`, qui contient la classe `StringTokenizer`, utilisée pour analyser les messages de demande HTTP. Considérant les lignes de code de la classe WebServer, il convient de définir deux objets de flux de données :

```
String requestMessageLine;
String fileName;
```

L'objet `requestMessageLine` est une chaîne qui contient la première ligne du message de demande HTTP. L'objet `fileName` est une chaîne contenant le nom du fichier sollicité. La série de commandes suivante est identique à la série équivalente du programme `TCPServer.java`.

```
ServerSocket listenSocket = new ServerSocket(6789);
Socket connectionSocket = listenSocket.accept();
BufferedReader inFromClient =
   new BufferedReader(new InputStreamReader
          (connectionSocket.getInputStream()));
   DataOutputStream outToClient =
      new DataOutputStream(connectionSocket.
            getOutputStream());
```

Ce passage conduit à la création de deux interfaces de connexion. Le premier, de type `Serversocket`, est un `listenSocket`. Généré par le logiciel serveur avant la réception d'une requête de connexion d'un client, il surveille l'accès 6789 en attente d'une requête d'établissement de connexion TCP. Lors de l'arrivée d'une requête, la méthode `accept()` de `ListenSocket` crée un nouvel objet, de type `Socket`, appelé `connectionSocket`. Puis deux flux de données sont générés : `BufferedReader` inFromClient et `DataOutputStream outToClient`. Le message de demande HTTP arrive depuis le réseau, traverse `connectionSocket` et rejoint le flux `inFromClient`. Le message de réponse HTTP rejoint le flux `outToClient`, traverse `connectionSocket` et

entre dans le réseau. En revanche, le reste du code diffère sensiblement de celui de TCPServer.java.

```
requestMessageLine = inFromClient.readLine();
```

Cette commande lit la première ligne du message de demande HTTP, qui est censée être de la forme suivante :

```
GET file_name HTTP/1.0
```

Notre serveur doit alors analyser la ligne pour en extraire le nom de fichier.

```
StringTokenizer tokenizedLine =
            new StringTokenizer(requestMessageLine);
if (tokenizedLine.nextToken().equals("GET")){
   fileName = tokenizedLine.nextToken();
   if (fileName.startsWith("/") == true )
      fileName = fileName.substring(1);
```

Les commandes précédentes analysent la première ligne du message de demande pour en retirer le nom de fichier sollicité. L'objet `tokenizedLine` peut être considéré comme la ligne de requête originale, dans laquelle les « mots » `GET`, `file_name` et `HTTP/1.0` sont placés dans différents emplacements, appelés jetons. Grâce au RFC relatif à HTTP, le serveur sait que le nom du fichier sollicité se trouve dans le jeton qui suit le jeton « GET ». Ce nom de fichier est inséré dans une chaîne de données appelée `fileName`. La fonction du dernier argument `if` dans ce passage de code est de supprimer la barre oblique pouvant précéder le nom du fichier.

```
FileInputStream inFile = new FileInputStream (fileName);
```

Cette commande associe un flux de données, `inFile`, au fichier `fileName`.

```
byte[] fileInBytes = new byte[numOfBytes];
inFile.read(fileInBytes);
```

Ces deux commandes évaluent la taille du fichier et constituent un tableau d'octets de taille équivalente, intitulé `fileInBytes`. La dernière commande s'étend du flux `inFile` jusqu'au tableau d'octets `fileInBytes`. Le programme doit manipuler des octets, car c'est le seul type de format de données que le flux sortant `outToClient` accepte.

Nous sommes maintenant capables d'élaborer le message de réponse HTTP. Pour cela, il nous faut tout d'abord insérer les lignes d'en-tête de réponse HTTP dans `DataOutputStream outToClient` :

```
outToClient.writeBytes("HTTP/1.0 200 Document
          Follows\r\n");
if (fileName.endsWith(".jpg"))
          outToClient.writeBytes("Content-Type:
          image/jpeg\r\n");
if (fileName.endsWith(".gif"))
          outToClient.writeBytes("Content-Type:
          image/gif\r\n");
```

```
outToClient.writeBytes("Content-Length: " + numOfBytes +
          "\r\n");
outToClient.writeBytes("\r\n");
```

Ce groupe de commandes est particulièrement intéressant. Ces commandes préparent les lignes d'en-tête du message de réponse HTTP et les transmettent au tampon TCP. La première d'entre elles envoie la ligne d'état obligatoire : `HTTP/1.0 200 Document Follows`, suivie d'un retour chariot et d'un saut de ligne. Les deux commandes suivantes préparent une ligne d'en-tête à contenu simple. Si le serveur doit par exemple transférer une image GIF, il formule la ligne d'en-tête `Content-Type:image/gif`. S'il s'agit d'une image JPEG, la ligne d'en-tête sera la suivante : `Content-Type: image/jpeg`. On suppose que notre serveur Web simplifié ne traite que les images GIF ou JPEG. Puis le serveur prépare et envoie une ligne d'en-tête fournissant des informations sur la taille du contenu, accompagnée de la ligne blanche qui doit impérativement précéder l'objet à envoyer. Il ne reste plus qu'à intégrer le fichier `FileName` au `DataOutputStream outToClient`.

Nous pouvons maintenant envoyer le fichier sollicité :

```
outToClient.write(fileInBytes, 0, numOfBytes);
```

Cette commande expédie le fichier, `fileInBytes`, en direction du tampon TCP. TCP concatène le fichier avec les lignes d'en-tête récemment créées. Il segmente le tout si nécessaire et envoie les différents segments TCP au client.

```
connectionSocket.close();
```

Après avoir terminé la requête, le serveur entame une phase de réorganisation au cours de laquelle il ferme l'interface `connectionSocket`.

Pour tester ce serveur Web, il suffit de l'installer sur un hôte quelconque, accompagné de quelques fichiers d'images GIF ou JPEG. Puis vous pouvez utiliser le navigateur de n'importe quel autre poste pour solliciter un fichier auprès du serveur. Dans ce but, vous devez utiliser le numéro d'accès que vous aurez incorporé dans le code (par exemple, 6789). Si votre serveur se trouve à l'adresse `somehost.somewhere.edu`, le nom du fichier sera `somefile.html`, et le numéro d'accès 6789. Le navigateur doit alors solliciter le fichier suivant :

```
http://somehost.somewhere.edu:6789/somefile.html
```

2.9 Distribution de contenu

La richesse du Web en volume de données et en contenu est énorme et croît chaque jour. Ce contenu se compose de pages Web, contenant du texte, des images, des applettes Java, des cadres HTML de toutes sortes, des fichiers MP3, des présentations audio, des films vidéo, etc. Ce contenu, réparti sur des milliers de serveurs aux quatre coins du monde, est également accessible de n'importe quel endroit du globe.